Transaction Level Platform Modeling in SystemC for Multi-Processor Designs

Lochi Yu, Samar Abdi, Daniel Gajski

Technical Report CECS-07-01
Jan. 25, 2007

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

lochi.yu@uci.edu, sabdi@ics.uci.edu, gajski@ics.uci.edu

Abstract

This report describes Transaction Level Platform Modeling in SystemC for MPSoC designs. The MPSoC platform is a net-list of processing elements, busses and bridge elements. The Processing Elements which can host a process (a C program) or memory. Busses, modeled as Universal Bus Channels (UBCs), offer communication functions for these processes and bridge elements (transducers) link different busses together. This platform yields an executable Transaction Level SystemC model, and has the advantage that the designer can use the existing C code and will yield a completely simulatable platform. To test the modeling style, 2 different platforms of a H264 decoder were developed and tested successfully. This report describes the internal structure of the busses, processing elements, and transducers of this model.

Contents

List of Figures

Transaction Level Platform Modeling in SystemC for Multi-Processor Designs

L. Yu, S. Abdi, D.Gajski
Center for Embedded Computer Systems
University of California, Irvine

Jan. 25, 2007

1 Introduction

Transaction level modeling using SystemC is emerging as a new paradigm for system modeling, since the rise of complexity, size and heterogeneity of modern embedded systems have raised the level of abstraction above RTL. On the other hand, platform based design [1] of multi processor SoCs (MPSoC) is beign adapted to combine the best features of top down and bottom up system design.

We present in this report a Transaction Level Platform Modeling style based in SystemC. In this platform we have C programs running inside Processing Elements (PEs), connected with busses which are linked by transducers. Each object in the platform is modeled according to a well defined SystemC template. Busses use a well-defined template called Universal Bus Channel (UBC) [2], transducers use their own defined General Transducer Architecture [3] template, processes are *sc_threads* and PEs are *sc_modules*. In the SystemC environment, a *sc_module* may have one or more *sc_threads*, and all *sc_thread*s run in parallel. Channels of communication, like the UBC, are defined as *sc_channel*s, while transducers are also *sc_module*s.

The platform is modeled as a top level *sc_module* which instantiates all UBCs, transducers, PEs, and connects all of them, as defined by the user in the GUI. This will create a fully executable TLM SystemC code which allows the designer to quickly simulate the platform in this high level abstraction level.

Related work TLM design in SystemC has gathered a lot of attention since it was introduced [4]. Several models and design flows [5] have been presented centering around TLM.

Other tools have been designed to facilitate platform designs: Metropolis [6] Platform-based design allows modeling of heterogeneous systems.

This report will describe the Processing Elements in section 2, the Universal Bus Channel structure in section 3, and the General Transducer structure in section 4. The platform model is shown

with two examples of a H264 decoder in section 5. Section 6 has the conclusions and future work.

2 Processing Elements

Every Processing Element (PE) can have processes and/or memory elements. We can define multiple PEs in a platform, and they must be connected to a bus. The processing elements that contains processes have a defined internal structure, which contains C code, global functions prototypes, and SystemC code.

2.1 Processes

The processes are the C programs that we want to run. These programs need to interface with SystemC code in order to do any communication tasks. Figure 1 shows how the code is organized in a process object.

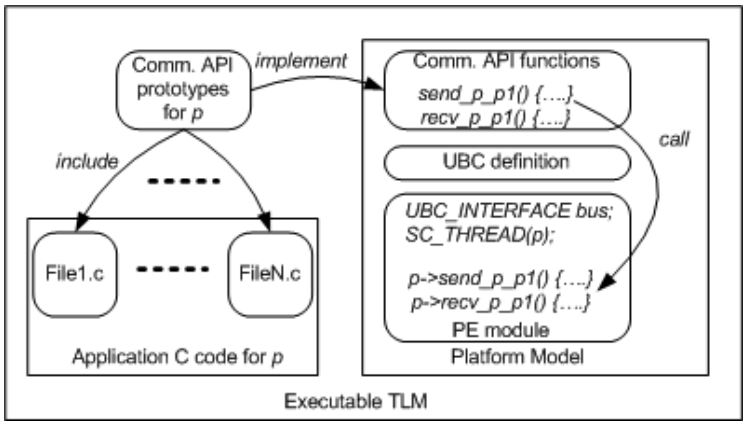

Figure 1: Executable TLM code organization.

We see in Figure 1 a representation of the executable TLM, which has 3 basic parts. First, there are the Communication API prototypes, which are the global functions prototypes that are included in the application C code (lower left corner). This code uses the Communication APIs to access the third block: the Platform model (right side of the figure). The platform model contains the Communication API code that accesses SystemC code in each PE module, in order to communicate with the busses.

A sample SystemC code for a process is shown below:

```
extern "C" int Intra(void);
void *ptr_Intra;

class P_Intra: public sc_module{
```

```
5    public:
         SC_HAS_PROCESS(P_Intra);
         P_Intra(sc_module_name name):sc_module(name){
                 SC_THREAD(main);
         }
10       sc_port<i_ubc> busport;
         int main(){
                 ptr_Intra=this;
                 Intra();
         }
15 };
```

Each process will reside inside a function in the SystemC class *sc_module*. The constructor will initialize all processes by defining the functions as independent *sc_thread*s (line 8). Line 10 defines a *sc_port* which will be the interface to the UBC. The communication APIs will access this port to communicate with the bus.

Inside each thread, a global pointer (declared in line 2) is assigned to the present object, and then the C program is finally called.

The communication APIs exported to the application C code are global functions which call the UBC methods inside the corresponding process' *sc_thread*. They are defined after each *sc_module*, one set for each process that communicates with the present one:

```
extern "C" void recv_P_ID_Intra_P_ID_Main(void *ptr, int size, int mode){
        P_Intra *p = (P_Intra*) ptr_Intra;
        unsigned int src= P_ID_Main;
        unsigned int dest= P_ID_Intra;
5       //Send request to transducer
        unsigned int r=size;
        p->busport->write(P_ID_Intra,ADDR_Intra_RECV_Main,
                (unsigned char*)&r,sizeof(unsigned int));
        p->busport->recv(P_ID_Intra,P_ID_Tx2,ptr,size,mode,&src,&dest);
10 }
extern "C" void send_P_ID_Intra_P_ID_Main(void *ptr, int size,int mode){
        P_Intra *p = (P_Intra *) ptr_Intra;
        //Send request to transducer
        unsigned int r=size;
15      p->busport->write(P_ID_Intra,ADDR_Intra_SEND_Main,
                (unsigned char*)&r,sizeof(unsigned int));
        p->busport->send(P_ID_Intra,P_ID_Tx2,ptr,size,mode, P_ID_Intra, P_ID_Main);
}
```

The pointer *p* in lines 2 and 12 refer to their specific PE; this was necessary since the global functions need to refer to one (and only) object of the process *sc_module*. This way, the C program will call these global functions and interface with the SystemC counterpart and access the busses' communication functions. Therefore, we must have one class per processing element, and only one object per class.

In summary, every Processing Element will be an *sc_module* with one or more *sc_thread*s, each one running C code. For every PE, there will be glocal functions(called by the C code) which will access the UBC communication functions.

2.2 Memory elements

In case of memory elements, what the *sc_module* contains is an array of variables and a port to communicate with the busses. Other PEs will write and read this memory using the UBC's communication functions.

3 Universal Bus Channel

This model abstracts the system bus as a single unit of communication. It provides the basic communication services of synchronization, arbitration and data transfer that are part of a transaction. At the transaction level, we are do not distinguish between different bus protocols. The bus is modeled as a *sc_channel*, implementing a *sc_interface* which provides 5 public bus communication functions:

1. *Send/Recv* for synchronized communication.
2. *Read/Write* for memory access.
3. *MemoryAccess* for memory control.

There are also 2 private functions, used by the above functions:

1. *ArbiterRequest/ArbiterRelease* for mutual exclusion.
2. *Synchronize* for synchronization.

In the present model, UBCs can only be connected to Processing Elements and transducers.

3.1 Synchronization

Synchronization is required for two processes to exchange data reliably. A sender process must wait until the receiver process is ready, and vice versa. A Synchronization Table in the UBC keeps the flags and events (indexed by process ids) that are used by a process to notify its transaction partner process that it is ready. Synchronization between two processes takes place by one process setting the flag and the other process checking and resetting the flag. Once the flag has been reset, the transacting processes are said to be synchronized. We will refer to the process setting the flag as the *initiator* and the process resetting the flag as *resetter*. The initiator and resetter processes for a given transaction are determined at compile time. In Figure 2 , assume P1 is the initiator process

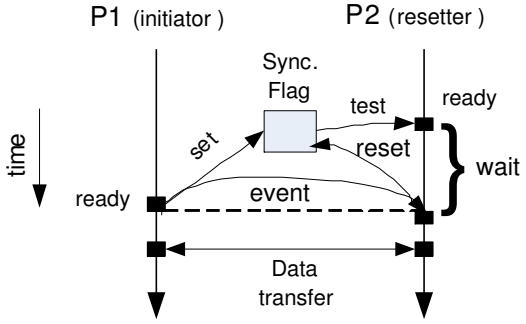

Figure 2: Flag-based synchronization between processes

and P2 as the resetter process. Hence, P1 sets the synchronization flag. If P2 is ready before P1, it must keep reading the flag until P1 sets it. P1 notifies this event when it sets the synchronization flag. Once P2 reads the flag as set, it recognizes that P1 is ready and resets the flag.

3.1.1 Implementation

The UBC model will have one flag and one *sc_event* for each pair of communicating processes. The synchronization by the two processes using Send/Recv functions is achieved by both calling the *Synchronize* function, which does one of two things, depending if the calling process is the initiator or the resetter:

```
unsigned int Synchronize(unsigned int MyID, unsigned int PartnerID,
   unsigned int MyMode) {
      if (MyMode==UBC_INITIATOR && MyID==P_ID_Tx2 && PartnerID==P_ID_Intra) {
         sync_Tx2_Intra =1;
 5       ev_sync_Tx2_Intra.notify();
         return UBC_INITIATOR;
      }
      if (MyMode==UBC_RESETTER && PartnerID==P_ID_Tx2 && MyID==P_ID_Intra) {
         while(sync_Tx2_Intra != 1){
10          wait(ev_sync_Tx2_Intra);
         }
         sync_Tx2_Intra=0;
         return UBC_RESETTER;
       }
15     ...
```

3.2 Arbitration

After synchronization, the resetter process will attempt to reserve the bus for data transfer. This is necessary since the bus is a shared resource and multiple transactions attempted at the same time

must be ordered sequentially. The resetter process will *request* an arbitration to the bus, and since the UBC model is exclusive for functional verification, the arbiter is modeled as a mutex (which is a *sc_mutex* in SystemC. An arbitration request corresponds to a mutex lock operation and once the transaction is complete, the process will *release* the arbitration with a mutex unlock operation.

3.3 Addressing and data transfer

In order to do addressing and data transfer, the UBC uses the following variables and events:

1. Variable *BusAddress* that stores the starting address of the active transaction;
2. Event *AddrSet* that is notified when *TxAddress* is set (it is implemented as a *sc_event*);
3. Variable *DataPtr* that keeps the pointer to the transacted data;
4. Variable *DataSize* that keeps the size in bytes of the transacted data;
5. Variable *RdWr* that identifies if a transaction is read or write (for Read/Write functions).

For synchronized communication, the *resetter* process sets *BusAddress* to the appropriate value from the bus address table. This is done by checking the process IDs and assigning the corresponding bus address:

```
if(MyProcID==P_ID_Intra && SendProcID==P_ID_Tx2)
        BusAddress=ADDR_DH_Tx2_Intra;
else if(MyProcID==P_ID_Trans && SendProcID==P_ID_Tx2)
        BusAddress=ADDR_DH_Tx2_Trans;
```

For memory transactions, the reader or write process sets *BusAddress*. This is followed by the notification of event *AddrSet* that wakes up the other process or memory controller that is snooping the address bus:

```
if(MyProcID==P_ID_Tx2 && SendProcID==P_ID_Intra){
        while(BusAddress!=ADDR_DH_Intra_Tx2){
                wait(AddrSet);
        }
}
```

In case of memory transaction, the memory controller reads the address *BusAddress* to check if the address falls in its range and computes the offset. If it is a *read* it sets *DataPtr* to the right address in the local memory according to computed offset, and if it's a *write*, it will proceed with the memory copy:

```
void MemoryAccess (unsigned int MEMLOW, unsigned int MEMHIGH, unsigned char *local_mem) {
   while (1) {  // memory is always servicing
      while (BusAddress < MEMLOW || BusAddress > MEMHIGH) {
         wait (AddrSet);          // every time some address is set
```

```
5       }
        if (RdWr == UBC_READ) {    // I am addressed for read operation
           DataPtr = local_mem + (BusAddress - MEMLOW);  // base + offset
           wait (SETUP_DELAY, SC_NS); // only for simulation
           wait (HOLD_DELAY+1, SC_NS); // only for simulation
10      }
        else if (RdWr == UBC_WRITE){       // I am addressed for write operation
           memcpy (local_mem + (BusAddress - MEMLOW),DataPtr, DataSize);
           wait (HOLD_DELAY+1, SC_NS); // only for simulation
        }
15   } // elihw (1)
} // end of MemoryAccess method
```

4 Transducer Model

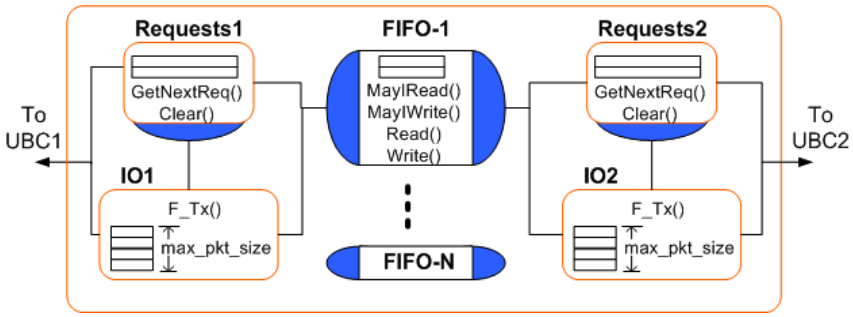

Figure 3: TLM for transducer module

The transducer connects two busses, and its purpose is to facilitate multi-hop transactions, where one process sends data to another process that is not directly connected to the sender via an UBC. The basic functionality of the transducer is to simply receive data from the sender process, store it locally and send it to the receiver process once the latter becomes ready. The transducer is modeled as a *sc_module* and there are three types of objects instantiated under the top level of the transducer.

4.1 Buffers

The data in transit via the transducer is stored in circular buffers, modeled as FIFO channels. The number of channels in a buffer is equal to the total number of communication paths through the transducer. Each buffer is modeled as a *sc_channel* and implements a *sc_interface* which supports four functions as follows:

1. *MayIWrite* returns true if the requested space is available in the buffer else returns false;

2. *MayIRead* returns true if the requested number of bytes are present in the buffer else returns false;

3. *BufferWrite* copies the incoming data to the buffer and updates the tail pointer;

4. *BufferRead* copies data from the buffer to the output and updates the head pointer;

4.2 Request Buffers

In general, before any data is sent/received to/from the transducer, a request must be made such that the transducer interface may check if the internal buffers can accomodate the data or supply it. Such a request may be included in the packet itself, but if the packet cannot fit, additional logic is needed in the bridge to reject the packet and in the process to check for rejection and resend it. For simplicity, we will only consider the scenario where the PE writes the request, followed by synchronization and data transfer. In case of multiple competing processes, the requests from different processes are arbitrated by the transducer and the communication with the successful process is initiated.

There are two request buffers in the transducer, one for each bus interface. The number of words per request buffer is equal to the number of communication paths through the bridge. The request buffer is modeled as any other memory module in a PE and thus has an address range on the bus. Each word in the request buffer has a unique bus address. The requesting process writes the number of bytes it expects to read/write into the communication path's corresponding request buffer. The request buffer is a module that supports two functions:

1. *GetNextReady* checks the request words in the buffer in a round-robin fashion. For the chosen request, it checks if the corresponding buffer has enough data/space to complete the transaction of requested size, calling the buffers' functions *MayIWrite* and *MayIRead*. If it returns *Tx_Yes*, it returns the request ID and path, else it checks the next pending request:

```
        ...
        if (RequestBuffer[1]) {
                *Near = P_ID_Trans;
                *Remote = P_ID_Main;
 5              *size = RequestBuffer[1];
                *TransferType = UBC_RECV;
                *Mode = UBC_RESETTER;
                if (OPB2DH->MayIRead(*Remote,*Near,*size) == Tx_Yes)
                        return true;
10      }
        if (RequestBuffer[2]) {
                *Near = P_ID_Intra;
                *Remote = P_ID_Main;
                *size = RequestBuffer[2];
15              *TransferType = UBC_SEND;
                *Mode = UBC_RESETTER;
                if (DH2OPB->MayIWrite(*Near,*Remote,*size) == Tx_Yes)
```

```
                return true;
        }
20      ...
```

2. *ClearRequest* removes the request from the buffer by setting the size to zero.

4.3 IO module

The IO module is the interface function of the transducer that talks to other processes on the bus. It starts by calling the *GetNextReady* function in the request buffer. Then, for the selected sender or receiver process, it calls the UBC receive or send function respectively. The IO module assumes the role of the Resetter if the process is the Initiator, and vice versa. The data received from sender is written to the corresponding FIFO. The data to be sent to the receiver is first read from the corresponding FIFO before calling the transducer send function. Once the requested transaction is completed, the request removed by calling the *Clear* function in the request buffer module.

5 H264 decoder models

In order to test our platform modeling style, we chose a H264 decoder and used the templates described above. The H264 decoder takes a .cif.264 file, decodes and displays it in the screen, using the Simple DirectMedia Layer (SDL) libraries in the GNU/Linux OS. We divided the decoder into 6 blocks:

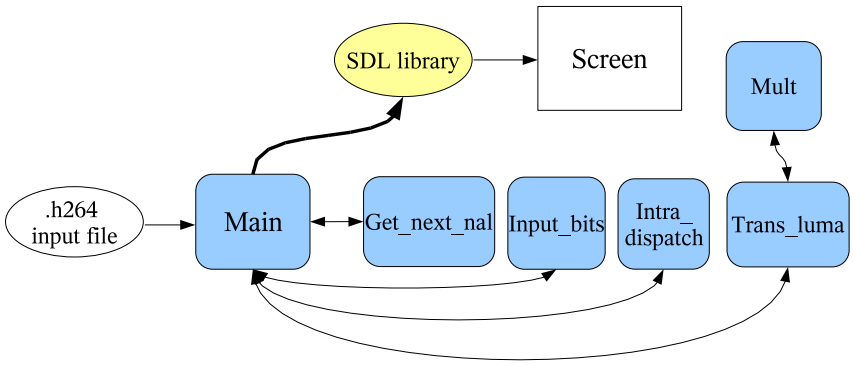

Figure 4: H264 decoder platform

The input file is a .cif.264 file with a video clip of at least 25 frames. It is read by the ”Main” block which does much of the processing. The main block initially calls the ”Get_next_nal” block which reads the file and obtains a ”NAL” unit, which is the logical data packet of the H264 video codec. The ”Input_bits” block manages the pointers of the NAL units buffer. The two other blocks

”Intra_dispatch” and ”Trans_luma” process the video in terms of their luma and chroma data. The last block ”Mult” does array multiplication for ”Trans_luma”.

Two different models were developed: a point-to-point model with 5 busses and a shared bus model with 2 busses and 1 transducer. The goal was to test the platform modeling style with two different platforms, by reading a set of input files and visualizing the output.

5.1 Point-to-Point model

This model shown in Figure 5 has one main *sc_module* with the main *sc_thread*. All other PEs communicate with the main module using an individual bus; there are a total of 5 busses. The functions were mapped one-to-one to each block.

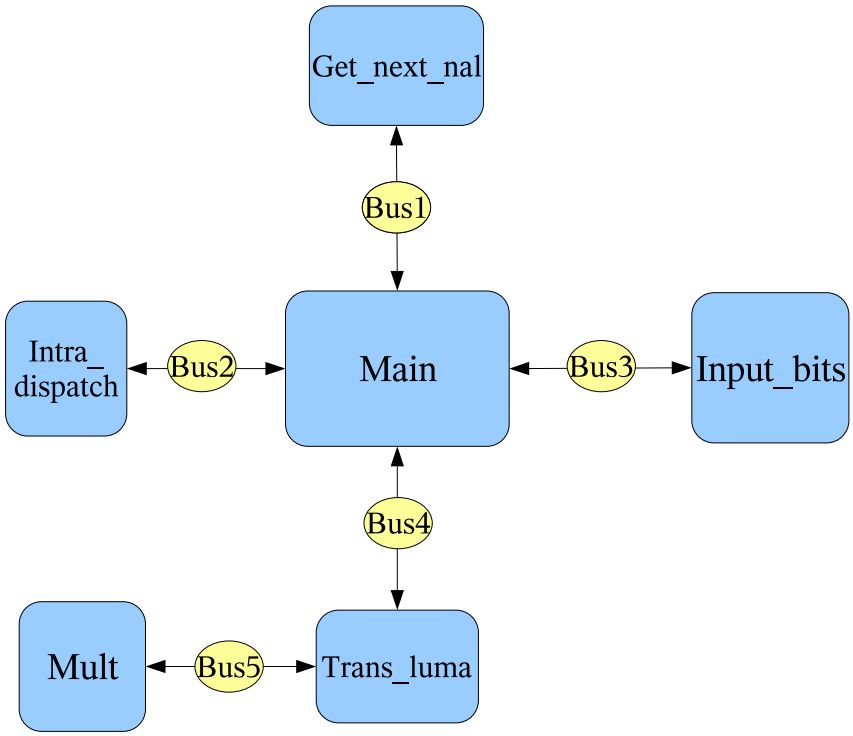

Figure 5: Point to point H264 decoder model

5.2 Shared bus model

The second platform also maps each function to a hardware block. The blocks share 2 busses and uses a transducer. This model shown in Figure 6. The main *sc_thread* sends and receives data

Table 1: Simulation times and code size for input clip of 25 frames

Results		
Platform	SystemC LOC	Sim time
C model	-	1.865s
Point-to-Point	1362	4.690s
Shared bus	1571	14.551s

from the PEs connected to Bus1 and from the other PEs connected to Bus2, via the transducer. There is also local communication between the module *Trans_luma* and *Mult* which does not use the transducer but uses only Bus2.

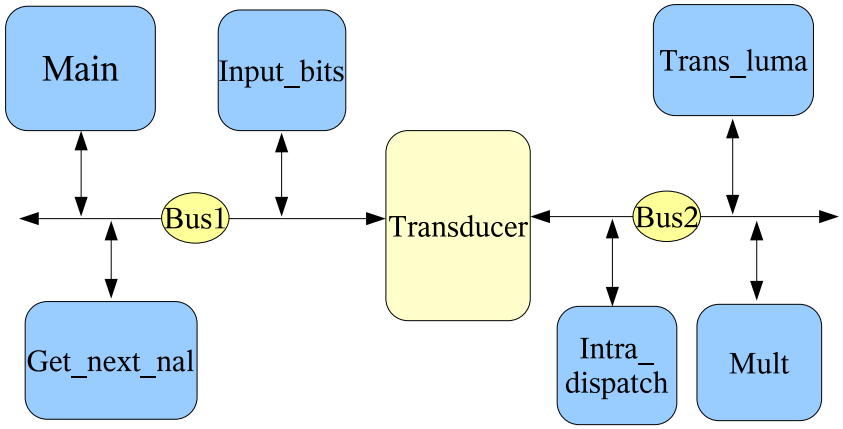

Figure 6: Shared bus H264 decoder model

5.3 Results

Both models were tested successfully with 4 .cif.264 files, with file sizes ranging from 28572 bytes to 632376 bytes. The frames decoded in each clip ranged from 25 frames in the smallest file to 360 frames in the largest. We present in Table 1 the simulation times for the smallest clip and code sizes of both models in comparison with the original C code of the H264 decoder. Note: the simulation environment was a Pentium 4, 2.80Ghz, 1Mb Cache, 1Gb RAM, running Linux kernel 2.6.9.

We can see in Table 1 that while the code size is similar between the point-to-point model and the shared-bus model, the simulation time is considerably higher in the latter one. In the second model, the arbitration of the busses comes into play since the two busses are not exclusive for any block, as was in the first model. Furthermore, the presence of the transducer and its FIFOs produce extra overhead that impacted on the simulation time.

6 Conclusions and future work

In this report, we presented a transaction level platform modeling style for MPSoC in SystemC. The model consists of processes, UBCs and transducers. The processes are implemented using *sc_thread*s inside PEs, which are modeled as *sc_modules*. Busses are modeled as *sc_channel*s and act as a single unit of communication, providing synchronization, arbitration and data transfer. The transducers connect two busses, and consist of buffers, 2 request buffers, and 2 IO modules. These models enable us to build an executable TLM in SystemC code, using the same C code to simulate the system. We tested the modeling style with 2 different platforms for a H264 decoder: a point-to-point model and a shared-bus model. Both performed successfully but with different simulation times. In the future, we plan to upgrade the transducer to allow transducer-to-transducer and transducer-to-memory module communication.

7 Acknowledgements

This work was supported in part by the Gigascale Systems Research Corporation (GSRC) under its Heterogeneous Systems Design pillar (Task 1.4.3.2). We would like to thank Abhijit Davare of UC Berkeley for pointing us to the original C reference of the H.264 decoder.

References

[1] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, A. Sangiovanni-Vicentelli. System Level Design: Orthogonalization of Concerns and Platform-Based Design. In *IEEE Transactions on Computer-Aided Design*, Vol. 19, No. 12, December 2000.

[2] S. Abdi, D. Gajski. *UBC: A Universal Bus Channel for Transaction Level Modeling*. Technical Report CECS-06-07, University of California, Irvine, April 2006.

[3] D. Gajski, H. Cho, S. Abdi. *General Transducer Architecture*. Technical Report CECS-05-08, University of California, Irvine, August 2005.

[4] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[5] A. Donlin. Transaction level modeling: flows and use models. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software co design and system synthesis*, pages 75-80, New York, NY, USA, 2004. ACM Press.

[6] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vicentelli. *Metropolis: An Integrated Electronic System Design Environment*. IEEE Computer Society, April 2003.